COSC 460
Honours Project Report
Department of Computer Science
University of Canterbury

ECSTASY

An Object Oriented Graphics System

Supervisor: Dr. W. Kreutzer.

Kerry Holling

October 1983

Table of Contents

1. Introduction

The purpose of this report is to describe the design and implementation of a simple two dimensional drawing system for use from within the Computer Science Department's DG SIMULA implementation. This includes a description of the capabilities provided by the system, its limitations and scope for future development. Appropriate user documentation, source code listings and graphics examples are given as appendices. They provide a users as well as a designers perspective on the graphics package and these will be referenced throughout. The system described in this report has been dubbed the ECSTASY Graphics System and shall be referred to as such from here onwards.

The main tasks of a graphics system such as this are to provide facilities for the construction of object representations and the generation of pictures from their descriptions. This also includes the ability to transform and manipulate such objects and the pictures which they form. The major design decisions thus centre around the internal representation of the graphics objects in the computer system and the transformation of that data into a corresponding picture on a display surface.

Section 2 gives a brief description of the hardware that was available for the development of the ECSTASY system. Section 3 centres on the class **plot_environment** (see the source code listing in appendix II) which contains the basic kernel of graphics routines necessary to implement such a system. These routines are based on the Core Graphics System (see reference [2]) and allow the user to draw pictures without concern for internal picture representation or output device characteristics.

The remainder of the ECSTASY software is to be found in class **animation** (see appendix II) which is discussed in section 4. This class contains the facilities which enable a user to define and manipulate graphics objects at an abstract level so leaving the details of internal object representation, manipulation and picture construction to the system. It is the design decisions involved in these areas which are discussed in this section.

A discussion on the concerns involved in the production of the final picture displays via device dependent display routines is given in section 5. This is followed by a summary and conclusion, after which it is hoped the reader will have a genuine understanding of the underlying ideas embodied in the ECSTASY Graphics System.

2. Hardware

The Computer Science Department's SIMULA implementation is on a DG Eclipse S/130 Computer. The peripheral devices connected to this machine include a Televideo 920 terminal, a DG Dasher printer and a nine track tape drive.

Neither the printer nor the terminal are suitable for high quality graphics display and this prevents any adequate development of an interactive graphics environment. None-the-less, the ECSTASY graphics system described here has been designed with an interactive application in mind in anticipation of the availability of a graphics terminal at some stage in the future. It should be easily interfaced into a complete, interactive graphics environment with a minimum of difficulty.

The only readily accessible, good quality plotter available during the course of this project was the Computer Centre's Printronix printer. When in plot mode this printer has a resolution of 792 by 792 overlapping dots per page and is consequently the device around which this project has been built.

Due to the nature of the Printronix the orientation with respect to picture production has centred on the construction of a bit map to represent dot matrix devices. Another example of this is the Televideo terminal which has been used as a 79 by 21 sized dot matrix (the number of columns by the number of rows) in order to facilitate the debugging of system routines with the display of crude, but none-the-less discernible, pictures.

The characteristics of the **Printronix** are such that each line of characters sent to the printer is treated on a dot row basis if the plot mode code (005 base 8) is sent either preceeding or following the string of plot data characters. Line feed commands associated with such rows will only advance the paper a single dot row space. Unfortunately, the printer cannot mix normal characters with plot data in the same line so making it necessary to print them under software control, hence the provision of the text facility discussed in section 3.2.2.

The low order six bits of every character sent on a plot data line represent six horizontal dots of the Printronix's matrix. It must be remembered, though, that the dots are printed in reverse order of this bit string representation, a fact reflected in the display routine for the Printronix (see appendix III).

3. Class Plot_Environment

The declarations and routines within this class constitute the basic kernel of drawing routines for the ECSTASY system. They are based on the specifications provided by the Core Graphics System (see reference [2]) with appropriate additions and alterations.

3.1 The Bit Map

The type of output devices towards which the ECSTASY graphics system is oriented are those that have a dot matrix representation. This design decision is naturally influenced by the properties of the main device available for use with the system, namely, the Printronix printer. Also, it is probable that this will be the predominant technology in future display devices and should thus be catered for.

In order to produce plot files for dot matrix oriented devices it makes sense, then, to have some internal representation for this matrix within the computer; an application for which a bit map is ideally suited. Each bit in the map can then have an 'on' and an 'off' representation, the status of which depends on whether the corresponding point on the display surface of the output device is to be made visible or not. The system drawing routines need only then affect the bits of this map, which at the end of a session can be dumped to a file for interpretation by a device dependent display routine.

It is, of course, desirable for any graphics system to be entirely device independent and to this end any dot matrix device can be made known to the system in order to have output produced for it. By knowing the size of each device, SIMULA's dynamic array declaration facility can then be used to declare a bit map of the appropriate size.

A minor problem was encountered in using this facility as an array declaration could not be made after a call to the initialisation procedure **get_device**. This procedure queries the user about the output device so that its size can be be determined. The problem was overcome by using a call to function **y_dev_sze** as the upper bound of the first subscript in the array declaration. Besides calling procedure **get_device** and returning the device size, this function also sets the default 'window', 'viewport' and character size settings for the system (see appendix I). This eliminates any need for an action section in class **plot_environment** apart from the declarations.

3.1.1 Internal Bit Map Representation

In theory, the most desirable implementation of a bit map would be a "packed array of boolean", but in SIMULA this is unfortunately not possible. The obvious space savings associated with such a representation should, however, be attempted to be emulated, especially when the small (64 KB) address space available to an ECLIPSE user is considered. It is for this reason that the bit map associated with the ECSTASY system is represented by a two dimensional array of integers, where the low order thirty bits of each integer represent thirty horizontal points of the dot matrix. The number of integers then needed to represent a horizontal row of the dot matrix is:

(number of dots per row - 1) div 30 + 1,

although it must be remembered that all the thirty bits in the last integer for the row may not be used.

The characteristics of the Printronix printer require the extraction of information from the bit map in six bit units (see section 2). This task is greatly simplified when a multiple of six bits per integer is used, hence the value thirty; the largest such multiple less than the total number of bits per integer.

While this decision was made with device characteristics in mind, little extra space is needed and calculation is made considerably simpler without violating device independence.

3.1.2 Storage Limitations

The Printronix printer needs a 792 by 792 bit map which, using the representation described above, requires 85536 bytes of storage. As a SIMULA user is restricted to a 64 KB address space it is obviously infeasible to represent this internally. Two solutions were considered for getting around this difficulty. The first was to store the bit map externally on disc, using a direct (random access) file, rather than in memory. However, the speed at which such a system would run would be unduly slow.

The only remaining option, then, was to decrease the size of the bit map and this is the course that has been taken. **The smallest plottable unit is thus considered to consist of a two by two square of plottable Printronix points.** The amount of storage required is immediately cut down to 21384 bytes, which is still large but certainly not unreasonable.

It should be noted that the size of the bit map for the Printronix has been further reduced from this 396 by 396 representation to one of 390 by 360 points, although it is for purely aesthetic reasons associated with the alignment of paper in the printer.

The design decisions associated with the need to extract information from the bit map in six bit units (discussed in section 3.1.1) must now, of course, hold for three bit units. Fortunately they do and no alterations need to be made, although the form of the display routine for the Printronix will be significantly different from what it otherwise would have been.

3.1.3 Implementation Overheads

Considerable extra overheads are involved with the bit map representation just described compared with, say, a "packed array of boolean" representation. This is made even more so with the lack of any logical `oring' and `anding' facilities on bit strings from within SIMULA.

In order to turn a particular bit `on', the integer in which it is located must first be identified. Its position in this integer must then be checked to ensure that it is not already `on', by a process of integer and modulo division, before the appropriate power of two can be added to the integer to effect the switching on of the bit. Similar manipulation of the integers is required by the display routines, discussed in section 5, when they interprete the integers after they have been dumped to a plot file.

3.2 Drawing Routines

To facilitate the drawing of pictures via the construction of a bit map a subset of the Core Graphics System developed by the ACM/SIGGRAPH Graphics Standards Planning Committee has been implemented. Functional descriptions of the implemented routines can be found in appendix I - The ECSTASY System Users Guide - as they are available to the user to construct pictures directly, without using the animation system facilities discussed in section 4. Extensive documentation of the Core Graphics System can also be found in reference [2], eliminating the need to discuss the routines in any more detail here other than to state what is provided:

(i) window
(ii) view_port
(iii) line_abs
(iv) line_rel
(v) mark_abs
(vi) mark_rel
(vii) move_abs
(viii) move_rel
(ix) reset
(x) send

The **reset** and **send** routines are not specific Core routines but are necessary for this application and are among those callable by the user.

The Core System also provides facilities for text production and the routines provided for this by the ECSTASY system are outlined in the Users Guide. Section 3.2.2 of this report is, however, also devoted to this subject because of the additional difficulties and concerns associated with implementing this feature.

3.2.1 Bit Map Manipulation

Additional to the Core System routines are those that are invoked by them to perform the actual bit map manipulation. These routines are at a lower level than the Core routines and are not callable by the user.

At the user level, all pictures are defined within a world coordinate domain and this must eventually be mapped to the device domain. The procedure **map_to_device** performs this function for a point. It takes the x and y coordinates of the point in the world domain as parameters, performs the necessary windowing and viewport transformations, and returns the device domain coordinates, albeit still in real units.

The drawing of lines at the device coordinate level is performed by the procedure **join** which in turn calls the lowest level routine in the system, procedure **plot.** This last routine truncates the real device coordinates given to it and turns on the appropriate bit in the map.

The truncation of the real device coordinates means that any device coordinate lying in the range [1, device_size+1) will be plotted. **No action is taken for points outside this domain, which are suppressed.** This windowing on a point basis has been chosen for its sheer simplicity and yet equal effectiveness over the alternative, windowing of lines.

3.2.2 Text

Because text and graphics cannot be adequately mixed on the Printronix printer there is a need to provide a text facility along the lines of that outlined in the Core Graphics System. The facility provided by the ECSTASY system is oriented towards the Printronix and provides facilities for text scaling and various directions of writing for all the printable characters. The ECSTASY System Users Guide (Appendix I) outlines these capabilities in greater detail.

Each character is represented by a sequence of drawing and moving instructions stored in the direct file **charcodes** (see appendix IV). An associated program **getchar** (listed in appendix IV) is used to input these instructions for each character. The use of this program and the meanings of the character codes are outlined in appendix I. It should be pointed out, however, that the character drawing instructions are in terms of Printronix device coordinates and so may not be applicable for other devices. This is especially true for those with a much higher or lower dot matrix resolution than the Printronix. The unit character size may then be either too small or too large to be considered suitable.

The aesthetic appeal of the characters produced by this text facility is not particular high, and actually deteriorates with each increase in text size. For unit sized characters the problem is largely due to the doubling up of the dimensions of plottable points as discussed in section 3.1.2. Thus, while the coded character representations are in terms of a seven by five dot matrix, the same as standard Printronix characters, they actually end up twice as high and wide as these standard characters without any improvement in resolution. The Printronix's overlapping dot feature, put to such good use for the standard characters, is also somewhat

negated by the doubling up of point sizes, resulting in sharper edges and less smooth slanting lines.

For all the character sizes the problems are further compounded by the inadequacy of our SIMULA implementation having to simulate floating point arithmetic because of the lack of a floating point unit on the ECLIPSE. The attempted addition of 1.0 to another floating point number actually results in 0.999 being added. This affects the length of lines drawn and is especially marked when text scaling is performed.

For example, the drawing of a line from (5.0, 6.0) to (5.0, 8.0) should result in device coordinates (5.0, 6.0), (5.0, 7.0) and (5.0, 8.0) being plotted, where an addition of 1.0 is made to the y coordinate in each step. The y coordinates actually calculated are 6.0, 6.999 and 7.998. This results in coordinate (5.0, 6.0) being plotted twice and (5.0, 7.0) once, so producing a line of length two rather than three.

This is a problem which can potentially affect all areas of the graphics system but no attempt has been made to try and `patch' this situation from within the system. Rather, the assembly routines used by SIMULA to perform floating point arithmetic need to be rewritten in order to perform this addition correctly.

4. Class Animation

Prefixed with class **plot_environment** this class provides the facilities for a user to define and manipulate pictures and the objects which constitute them at an abstract level.

4.1 Geometric Transformations

Three coordinate transformations have been implemented in the ECSTASY system. They are:

(i) Rotation
(ii) Translation
(iii) Scaling

By invoking the appropriate routines any object, including a complete picture, can be subjected to the above transformations to create a new, distinct object instance. The routines involved are all implemented as function procedures which return a reference to the new object, so leaving the original object unchanged. The reference variable for the original object can, of course, be assigned to the newly created instance to achieve the same result as a transformation applied directly to the original.

These three transformations are all bijective mappings from the two dimensional world coordinate space onto itself and preserve the given incidence relations between the points of an object or picture (e.g relations of the kind "one line connected to another etc.").

4.1.1 Rotation

Rotation within the XY-coordinate system requires three necessary items of information:

(i) the angle of rotation, counter-clockwise

(ii) the x coordinate of the point of rotation

(iii) the y coordinate of the point of rotation

The routines that perform this transformation on the various types of objects require this information to be passed to them via their parameters. The geometric relations between the original XY-system and the system X'Y' obtained after the rotation by the angle a are:

$$x' = x.\cos(a) + y.\sin(a)$$

$$y' = -x.\sin(a) + y.\cos(a)$$

4.1.2 Translation

Translation of a point is performed by adding a positive or negative constant to each coordinate of the point. The two parameters required by the translation routines are precisely these x and y values expressed in world coordinates.

4.1.3 Scaling

Scaling of a point is performed by the multiplication of a constant to each coordinate of the point. The x and y scaling factors are required as parameters to the routines which perform this transformation.

If the scaling factor in the x or y direction is less than one then the object generated from the scaling will be compressed in that direction compared with the original. Similarly, if the scaling factor is greater than one then the object will be stretched.

While not only creating an object of different size, scaling will usually cause the new object to be at a different position relative to the origin than the original. For example, scaling a point at position (10, 10) by a factor of two in the x and y directions will cause the new point to occur at position (20, 20). If the origin is an interior point of the object, however, this associated shift will not occur i.e. a point a (0, 0) will always give rise to another at this position.

4.2 Picture Representation

The most important design decision that has had to be made in the development of the ECSTASY system is the internal computer representation of complete pictures and the elements which comprise them. Every picture consists of one or more picture elements which in turn may consist of further distinct elements, down to the lowest, or primitive level. The chosen representation must reflect this hierarchical nature of picture structure.

The SIMULA programming language offers two facilities to aid in the construction of such hierarchical structures:

(i) class concatenation

(ii) SIMSET linked list facility

Both these facilities have been used to design a simple, yet effective and flexible, picture representation scheme.

At the top level of the hierarchy there is the need, using SIMULA's class mechanism, to enable reference variables of the appropriate type to refer to an object representing a complete picture. To this end the class type **picture**, prefixed by the predefined SIMSET class **head**, has been defined. This enables objects of this type to be the root of any hierarchical structure. Those elements which comprise a picture at the next level of abstraction can then be associated with the particular picture object through their membership of the linked list of which it is the head. This picture structure is illustrated in figure 1.

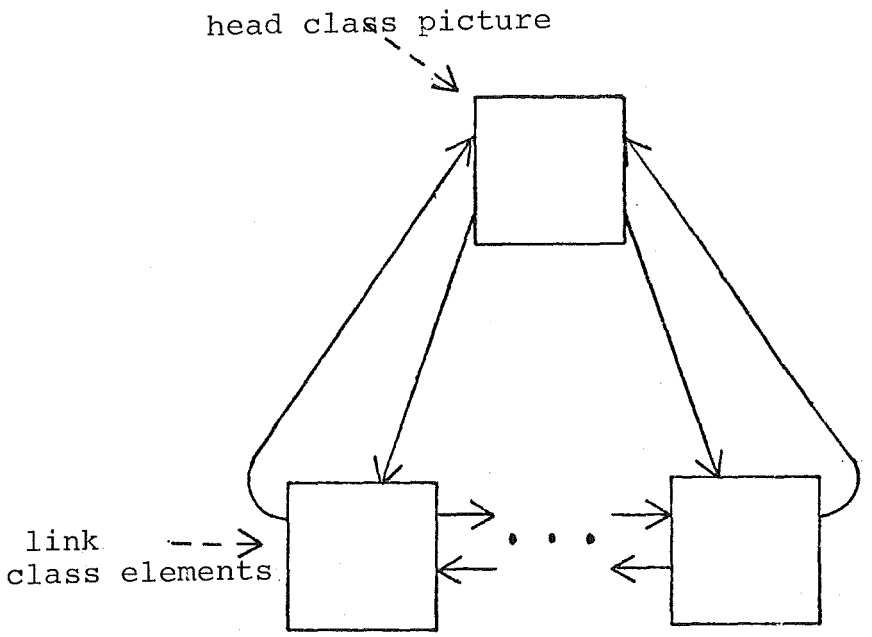


Figure 1. Internal Picture Representation.

The advantages of this scheme include the ability to have a variable number of elements comprising a picture without having to impose an upper bound on this number. Also, the provision of predefined SIMSET routines for inserting and deleting elements from structures of this kind removes the details involved in these operations from the graphics system itself.

This structure promotes localisation of declarations and code, as well as ease of picture manipulation and drawing, as will be illustrated in the following sections.

4.2.1 Drawing

The drawing of a complete picture simply involves accessing each element which comprises it and invoking the drawing routine local to this element. Section 4.3 will outline this in more detail.

4.2.2 Geometric Transformations

The structures of the routines to perform the available transformations are basically the same. A new picture object is first created to be the head of the newly generated picture. Each element comprising the original is then accessed and the transformation routine local to it invoked. The new element created from this procedure is then entered into the linked list for the new picture.

As a uniform transformation is applied to each element of the original picture, the new picture instance will represent a direct mapping under transformation of the original.

4.3 Picture Elements

So far, the element objects which comprise a picture have been referred to as if they were all of the same kind. This is, of course, not necessarily the case, as pictures in the ECSTASY system can be comprised of points, polygons, lines and circles. Needless to say, distinct class declarations are required for these various kinds of elements due to their differing representations and drawing and transformation requirements.

However, at the level of picture manipulation discussed in section 4.2 it is necessary to be able to refer to these distinct elements as if they were the same. This can be achieved by using SIMULA's class concatenation facility to prefix the class declarations of the various element types with a common super-class, namely, class **element**. This then enables reference variables with qualification element to refer to any instance of the various element objects.

Since all elements can be drawn, rotated etc. it is at the element level that these notions are introduced. However, the actual mechanics of these operations depend entirely on the nature of the sub-class and cannot, therefore, be defined at this level, hence the VIRTUAL procedure declarations within this class. Class element must, of course, also be prefixed by the predefined SIMSET class **link** to enable any element to become a member of a linked list associated with a picture.

The next sections of this report outline the representations and associated routines of the various element types that can comprise a picture.

4.3.1 Circles

A circle can be precisely defined in world coordinate space by the specification of a centre point and a radius. Also associated with objects of this type are two variables, x_scale_factor and y_scale_factor, the functions of which are discussed in section 4.3.1.2.

4.3.1.1 Drawing

When drawing a circle, each individual point on the circumference must have its position calculated in order for it to be marked on the bit map. To ensure that none of these points are missed out the angle used in calculating the position of successive points must be incremented by sufficiently small amounts each time.

This increment, however, need only be as precise as the output device is in representing curved lines, a feature dependent on the resolution of its dot matrix. Consequently, the expression for the increment involves the size of the dot matrix for the output device in the x direction (which is usually larger than in the y direction). Such a device dependent increment prevents the need for substantial extra calculation when constructing circles for the Televideo terminal, for example, compared with the Printronix printer.

4.3.1.2 Scaling

When scaling is applied to a circle with differing scale factors in the x and y directions the object created is an ellipse. The variables **x_scale_factor** and **y_scale_factor** associated with each circle object are consequently used to keep track of the amount of scaling done in the two directions. Then, when the drawing routine for the circle is invoked, these variables will be used to modify the positions of the points that would otherwise have been calculated as lying on the circumference of a circle rather than an ellipse.

It should be noted that these variables are always initialised to one at the creation of each new circle object but can be changed by the user in order to generate an ellipse rather than a circle. A severe limitation of this feature, however, is the inability of ellipses to be elongated in any directions other than horizontal or vertical. Even a rotation, while shifting the centre point of an ellipse, will not change its orientation.

4.3.1.3 Rotation and Translation

Both these geometric transformations applied to a circle will result in an object of the same dimensions (and orientation in the case of an ellipse) but with a different centre point from the original, determined by the transformation parameters.

4.3.2 Points

A point can be precisely defined by its x and y coordinates in the world domain. The geometric transformation routines can all be applied to the specified x and y coordinates as outlined in section 4.1.

4.3.3 Polygons

Any closed shape in two dimensions that consists of a set of concatenated vectors, or straight-line segments, is a polygon. An infinite number of shapes is covered by this description and any system designed to represent polygons must be flexible enough to cater for them all; an aim which can only be achieved by the implementation of a dynamic polygon representation.

This description of polygons is in terms of vectors; however, a vector can be defined by specifying only one point, its end point, whereas its start position is given by the current world coordinate position indicator (which may be at the end of the previously defined vector). The definition of a polygon is made much more economical and concise by simply specifying the world coordinate positions of its vertices (or vector end points) and this is the approach taken here.

What is required then is a dynamic structure for storing the variable number of points needed to adequately define any particular polygon; an application which lends itself to the use of the SIMSET facility in exactly the same way as used in representing pictures. The only differences in this case are the homogeneous nature of the elements that comprise the linked list, compared with a picture, and the use of a variable of type **ref (head)** that is local to the class declaration. This variable is necessary as the polygon class declaration is already prefixed by **element** and so cannot also be prefixed by **head**. The fact that polygons can be members of linked lists as well as giving rise to them is a perfect example of the hierarchical nature of picture structure. The structure of polygon objects is illustrated in figure 2.

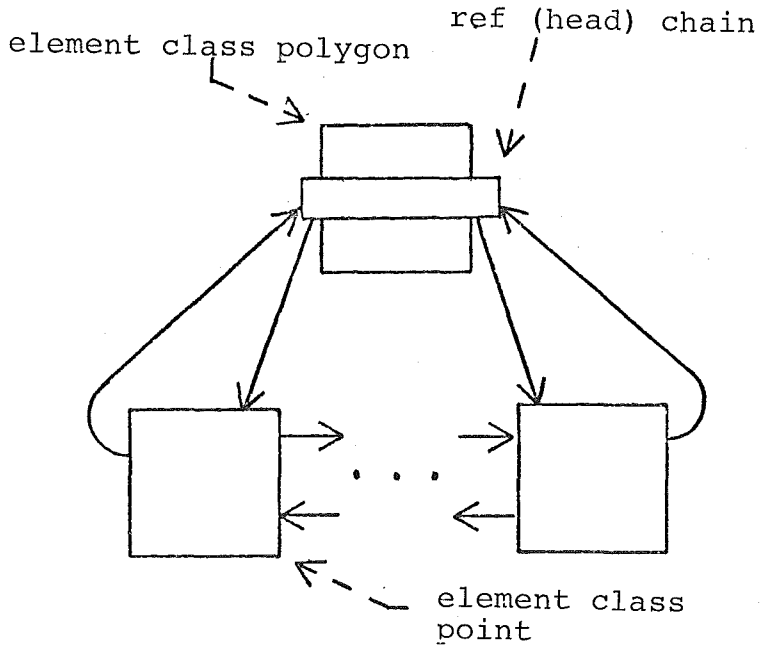


Figure 2. Internal Polygon Representation.

When an instance of a polygon is created it is initially undefined as no parameters are passed to it. This is because two methods of defining a polygon have been implemented and it is up to the user to invoke the appropriate routine for her application. This flexible approach allows the polygon to be defined via one call to procedure define or by multiple calls on procedure include. This latter method is useful, for example, when an unknown number of points are to be read from a file.

4.3.3.1 Drawing

Each polygon is drawn by first moving the current position indicator to the first point in the linked list. A line is then drawn to each successive point from the previous one until the last point in the linked list has been connected. A final line is then drawn from this last point back to the first, so closing the shape.

4.3.3.2 Geometric Transformations

The structures of the three routines provided to perform these transformations are identical. In each, a new polygon object is created. Then, for each point in the linked list of the original polygon, the appropriate transformation is applied and the newly created point is added to the linked list of the new polygon.

As a uniform transformation is applied to each point, in the same order as they appear in the original polygon's list, the effect is to produce a new polygon that is a direct mapping under transformation of the original.

4.3.4 Lines

A line, or straight line segment to be more correct, is considered to be a polygon defined by only two points, a start and an end. By prefixing class **polygon to the class declaration of** line all the routines discussed in the previous section can then be applied to objects of this type.

The fact that a specific, known number of points is needed to define a line, enables them to be passed as parameters to the class declaration and used to construct the appropriate representation.

5. Display Routines

Irrespective of which device the picture represented by the bit map is to be displayed on, the plot file produced by the graphics system must be interpreted by an independent, device dependent display program to perform the plotting. At present two such routines exist, one on the Eclipse for the Televideo and the other on the Prime for the Printronix.

5.1 Televideo Display Routine

The display routine for the Televideo (see appendix III) simply clears the screen and extracts the information bit by bit for each integer on the plot file, three integers per row for twenty-one rows.

A test must be done during this process to ensure that only nineteen of the thirty possible bits in the last integer for each line are used, so giving seventy-nine dots per row.

5.2 Printronix Display Routine

The steps that must be followed to produce a display on the Printronix from a plot file on the Eclipse are certainly not as straightforward as those for the Televideo.

The plot file must first be copied to magnetic tape using the **ETOP** command. This tape must then be taken to the Prime operators to be mounted so that the plot file can be copied to disc (A detailed summary of the commands involved is given in the ECSTASY System Users Guide). At this point the file is not in a suitable state for input to the display routine and must be preprocessed by the **ETOP** program listed in appendix III. The plot file produced by this program can then be used as data to the display routine, from which, a spoolable file is produced. The display routine itself is quite self documenting and is listed in appendix III.

5.3 Charging

The costs associated with the just described procedure for producing a display on the Printronix are quite prohibitive. At fifty cents to mount a tape and approximately fifteen cents to print over 700 lines (be it all on one page), an overhead of sixty-five cents is immediately incurred. This is without considering the processor time also required to produce the final spoolable plot file.

At present the only fast, immediate and cheap display that can be obtained is through using the Televideo 920. This is, however, less than adequate, except perhaps for debugging purposes, where it can sometimes be used to give a general idea of the form a picture will take.

6. Summary and Conclusions

There is obviously scope for the ECSTASY system to be further developed and extended and indeed it has been designed to facilitate this. The basic kernel of routines can be extended to include more of those outlined in the Core Graphics System. Similarly, new types of objects can be added by the user for specific applications and situations.

There are a large number of high level graphics facilities over and above those implemented here which can be added to the system over time. These include such things as the ability to blank out areas of the bit map enclosed by a particular object, so that objects may be overlayed, and the specification of point and line types to be drawn by the system i.e. cross, dot, x for points and dashed, solid or dotted for lines. There is also scope for the representation of solid objects in the three dimensions but this is considerably more difficult to implement than what has been achieved here.

More importantly, perhaps, there is great scope for the system to be interfaced into a complete, interactive environment. An aim which would come considerably closer with the availability of a good graphics terminal. Then, on top of the ECSTASY system as it stands, would be the need for a higher level, user interface package. Something which would undoubtedly be an implementation exercise in itself.

At the outset of this project the aim was to first implement facilities for the production of chronological graphs, histograms, pie charts etc. from files of simulation output data. This was to be followed by the attempted implementation of a drawing system based on geometrical primitives. After the basic kernel of drawing routines had been implemented, however, the possibilities of a basic drawing system such as the one described in this report became apparent and seemed more realistic to achieve. A change in priorities subsequently ensued to enable a more detailed investigation into this somewhat more general area than the production of data displays. Even so, the ECSTASY system provides the facilities to make the production of data displays considerably easier. A natural extension of the system would be to provide facilities for the automatic production of displays from files of output data, but time considerations have prevented this step from being taken at present.

It is hoped that the impression given by this report is of a flexible and easily extendible graphics environment framework, characterised by its simplicity and ability to be further developed without disrupting existing software. While the system is currently implemented in a totally non-interactive environment this should in no way detract from its appeal as a basis for an interactive system in the future.

7. References

[1] Birtwistle et al.
SIMULA Begin
Auerbach Publishers Inc.
Philadelphia, Pa., 1973.

[2] Denning P.J. (Editor)
ACM Computing Surveys
Special Issue: Graphics Standards
Vol. 10, Number 4, December 1978.

[3] Giloi W.K.
Interactive Computer Graphics -
data structures, algorithms, languages
Prentice-Hall Inc.
Englewood Cliffs, New Jersey, 1978.

APPENDIX I

ECSTASY System Users Guide

Table of Contents

1. Introduction

This document is a users guide to the ECSTASY Graphics System and should be read in conjunction with the report - ECSTASY: An Object Oriented Graphics System - of which this guide is also an appendix. As the routines provided by the system are to be called from within a SIMULA program a working knowledge of this language is assumed in the reading of this guide. The system is implemented from within the Computer Science Department's DG SIMULA implementation on an ECLIPSE S/130 computer.

The ECSTASY system provides facilities for the definition and manipulation of geometrical primitives at an abstract level without the user having to be concerned with actual internal object representations. These facilities are outlined in section 4. A set of basic drawing routines is also provided for the user to construct pictures directly without using these higher level facilities. These are outlined in section 5.

At present the only two output devices catered for by the system are the Televideo 920 terminal attached to the ECLIPSE and the Computer Centre's Printronix printer. The system produces plot files of integers for interpretation by device dependent display routines associated with these devices. The steps involved in producing the final displays are outlined in section 7.

2. User Program Interface

The source code for the system is contained in the file ECSTASY. This is incorporated into a user program in the following way:

```
begin
%include ecstasy
animation
   begin
      .
      .
   ( user program )
      .
      .
   end
end;
```

By prefixing the user program block with class 'animation' the facilities provided by the system are immediately available for use from within this block. Note also the necessary begin..end pair that surrounds the complete program.

3. System Initialisation

Immediately a user program begins executing with the ECSTASY system a query of the form:

```
Output devices:
  1: Printronix
  2: Televideo

Enter Your Choice:
>
```

will appear, to which either the response '1' or '2' must be typed. Any other character will cause the query to be repeat. All the plot files subsequently produced by the system will be for the specified device and this cannot be changed.

The next two sections outline the system routines for specifying the 'window' and 'viewport' settings. Both these must be specified before the construction of a new picture or the current settings, which may be the defaults, will be used.

3.1 Window(x_min, y_max, y_min, y_max, adjust);
real x_min, y_min, y_min, y_max;
boolean adjust;

A window is a rectangular domain in the world coordinate space. By specifying this domain with the above call, coordinates that are more natural for any particular application may be used.

The first four parameters define the low and high limits of the window along each coordinate axis. Knowing these picture frame limits the system will set new conversion factors for the transformation between the world and device coordinate domains.

This is done in such a way that for the case adjust=false the picture will not be drawn true to scale. The whole surface of the display device will be used for the mapping from world to device coordinates. This is useful, for example, when scattergrams with differing x and y coordinate ranges are required to be drawn using the whole of the display surface. Also, for the case adjust=false, the differing height to width ratios of plottable points on many particular devices is not adjusted for.

For the case adjust=true pictures will be drawn true to scale. This must be done, for example, to prevent circles from being drawn as ellipses.

The window settings may be changed at any stage of picture construction but care must be taken when doing this. It is usually only done in conjunction with a change in viewport as discussed in the next section.

The default window setting is precisely the device coordinate domain with adjust=false. i.e. for a terminal with 22 rows and 79 columns the default would be (0, 22, 0, 79, false).

3.2 Viewport(x_min, x_max, y_min, y_max);
real x_min, x_max, y_min, y_max;

A viewport is a rectangular portion of the display surface onto which the window is to be mapped. By default this is the whole of the display surface. The parameters are in terms of normalised device coordinates i.e. between 0 and 1. For example, viewport(0, 0.5, 0.5, 1.0), will cause the top left quarter of the screen to be used.

If a viewport is not of the same relative dimensions as the whole of the display surface then pictures that would otherwise have been drawn true to scale will again be distorted. Care must be taken in allowing for the interaction between various window and viewport settings.

At any stage in picture production the viewport setting may be changed in order for a different area of the display surface to be used. The default setting is (0, 1, 0, 1).

4. Drawings Based on Geometrical Primitives

The ECSTASY system provides facilities for the definition and manipulation of objects of the following types:

(i) complete pictures
(ii) circles
(iii) points
(iv) polygons
(v) lines

Object instances of a particular type must be referenced by a variable of the appropriate type, namely:

(i) ref (picture)
(ii) ref (circle)
(iii) ref (point)
(iv) ref (polygon)
(v) ref (line), respectively.

Each object can be drawn, rotated, translated or scaled. As the routine calls to perform these operations are the same for all objects they are discussed first before the distinct characteristics of the individual object types are outlined. The following discussion uses a general reference variable 'object' to represent an instance of either one of the possible object types.

The three geometric transformations are all implemented as function procedures and return a reference to a new, distinct object instance. This instance is a direct mapping under transformation of the object to which it was applied. By assigning the reference variable for the original object to the function call, however, the effect is the same as a transformation applied directly to the original.

i.e. object :- object.rotate(...).

A more detailed discussion on geometric transformations is given in the ECSTASY System report mentioned earlier.

(i) Object.draw;

This will result in the referenced object being displayed on the output device when the plot file produced by the system is interpreted by the appropriate display routine (see section 7).

(ii) Object.rotate(xcentre, ycentre, angle);
real xcentre, ycentre, angle;

The new object instance will be a rotation of the instance referenced by 'object' of angle **radians**, counterclockwise, about the point (xcentre, ycentre) in the world domain.

i.e. object :- object.rotate(100, 100, 3.1415926);

(iii) Object.translate(x_dir, y_dir);
real x_dir, y_dir;

The new object instance will be a translation of the instance referenced by 'object' of x_dir world coordinate units in the x direction and y_dir units in the y direction.
i.e. another_object :- object.translate(10 , 20);
(where another_object is of the same type as object)

(iv) Object.scale(xscale, yscale);
real xscale, yscale;

The new object instance will be a scaling of the instance referenced by 'object'. If the scaling factor in the x or y direction is greater than one then the new object will be stretched in that direction. If it is less than one then the object will be compressed.

4.1 Picture Objects

A picture object is made up of any number of other object instances (except other pictures) to form a complete picture.

(i) pix :- new picture;

- creates a new instance of a picture referenced by 'pix'. Initially no other objects are associated with the new picture.

(ii) pix.include(object);

- adds the object instance referenced by 'object' to the picture 'pix'. If this instance is already in another picture then it will be removed from it before insertion into 'pix'. Any particular instance can only be in one picture at a time.

(iii) object.out;

- will remove 'object' from its present picture, if any.

4.2 Circle Objects

```
(i) circ :- new circle( pt, radius );
            ref ( point ) pt;
            real radius;
```

- creates a new object instance that represents a circle with a centre at 'pt' and a radius of 'radius' world coordinate units.

```
(ii) circ.x_scale_factor := 2, or
     circ.y_scale_factor := 0.5.
```

- will change the scaling factor in the specified direction thus elongating or shortening the dimensions of the circle in that direction. This enables ellipses to be defined although the elongation can only occur in the x and y directions.

4.3 Line Objects

```
(i) lin :- new line( x1, y1, x2, y1);
           real x1, y1, x2, y2;
```

- creates a new object instance representing a line from (x1, y1) to (x2, y2) in world coordinate space.

4.4 Point objects

```
(i) pt :- new point( x, y );
          real x, y;
```

- creates a new object instance representing a point at world coordinates (x, y).

4.5 Polygon Objects

A polygon consists of a set of points which represent its vertices. The order in which these vertices are connected is the same as that in which they were entered into the polygon representation.

(i) poly :- new polygon;

- creates a new polygon instance which has yet to have its shape defined. The following two routines are provided for this.

```
(ii) poly.define( vertices, number );
          real array vertices;
          integer number;
```

The dimensions of the array 'vertices' must be at least (1:2,1:number). Each pair of points, vertices(1,k) and vertices(2,k), is used to create a new point object which is then added to the list of vertices for the polygon. The number of vertices is specified by 'number' and they must be in the order that they are to be connected when the polygon is drawn.

```
(iii) poly.include( pt );
           ref point (pt);
```

- is used to insert the points representing the vertices of a polygon on an individual basis. This is useful if they are being read from a file as happens in the routine outlined in the next section.

4.6 Get_Objects Facility

The routine:

```
get_objects( filnam, number, pix );
    text filnam;
    integer number;
    ref ( picture ) pix;
```

will read data pertaining to 'number' objects from file 'filnam'. Each line of this file must take one of the following forms depending on which type of object is represented. The first letter must be in column one.

(i) circles

C <x centre> <y centre> <radius>

(ii) lines

L <x1 coord> <y1 coord> <x2 coord> <y2 coord>

(iii) points

P <x coord> <y coord>

(iv) polygons

G <number of vertices> (<x vertix coord> <y vertix coord>)+

5. Basic Drawing Routines

As well as the facilities discussed in the previous section there is a basic set of drawing routines that can also be used to construct pictures. These routines are a subset of the Core Graphics System (see ACM Computing Surveys, Volume 10, Number 4, 1978) and enable the drawing of the basic output primitives (i.e lines, points and text) under direct control of the user program. A number of the routines make use of the current position indicator (CP) which is maintained by the system. This indicator takes on values corresponding to the current drawing location in world coordinate space and its use reduces the number of arguments required by many of the routines.

A description of the routines now follows. All coordinates are expressed in real values.

(i) line_abs(x, y);

- draws a line from the current position to the world coordinates (x, y). CP becomes (x, y).

(ii) line_rel(x, y);

- draws a line from the current position to the coordinates (x, y) relative to the current position. CP becomes this new position.

(iii) mark_abs(x, y);

- draws a point at coordinates (x, y). CP becomes (x, y).

(iv) mark_rel(x, y);

- marks a point at coordinates (x, y) relative to the CP. CP becomes this new position.

(v) move_abs(x, y);

- CP becomes (x, y).

(vi) move_rel(x, y);

- CP becomes the point displaced (x, y) from the current position.

(vii) reset;

This is not a specific Core routine. It clears the internal picture representation of all previous drawings so that a new picture can be started.

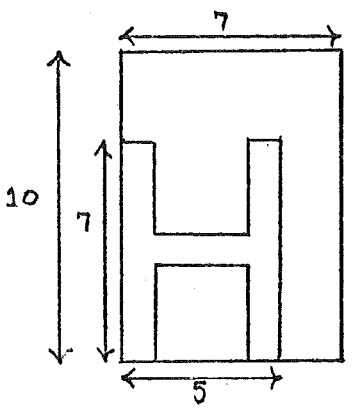
(viii) send;

Also not a specific core routine, send will dump the contents of the internal picture representation to a plot file for later interpretation by a display routine. The user is prompted for the name of this file. The display routines are discussed in more detail in section 7.

```
(ix) prOtext( line, xstart, ystart,
              x_char_space, y_char_space );
                 text line;
                 real xstart, ystart,
                      x_char_space, y_char_space;
```

This routine will cause the text string referenced by 'line' to be displayed starting at world coordinate position (xstart, ystart). The bottom left of the first character is taken to be at this point.

Each character is defined within a 7 by 5 dot matrix and is drawn within its own 10 by 7 square of device coordinates. For example, the letter 'H' is:



Care must be taken that the characters expressed in device coordinates do not overlap with other picture objects expressed in the world domain. The user should experiment with each output device to check the size of the characters produced.

The routine for entering the coded character representations used by the system is discussed in the next section. It enables the layout for any character to be easily changed within the bounds of its 10 by 7 matrix.

The x_char_space and y_char_space parameters determine the direction the text will be written in. They are expressed in character position units i.e.

```
( 1,  0 )    is horizontal across,
( 1,  1 )    is diagonal up to the right,
( 0, -1 )    is vertical down,
( 0, -1.5)   is vertically up, one and a half spaced.
```

```
(x) text_size(x, y);
      integer x, y;
```

This routine is used to change the text size attributes in the x and y directions. The default is (1, 1). For example,

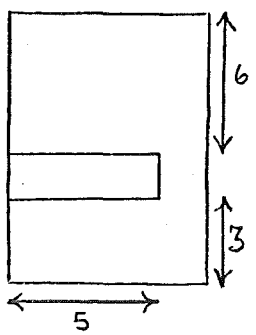
(2, 2) is double sized characters,
(1, 2) is double height by single width.

Again, the user should experiment with the output device to check the size and quality of the characters produced with various combinations of height and width sizes.

6. Coded Character Representation

Each character is represented by a series of moving and drawing instructions stored in file **charcodes**. The program **getchar** can be used to change these representations and is easy to use, prompting the user at every stage. Each drawing or moving instruction is represented by three integers. The first integer is either 0 (for draw) or 1 (for move) and is followed by the x and y displacements for the operation from the current position. The last instruction, which may not always be needed, must be a move to ensure the current position is returned to the bottom left of the character matrix. It can be assumed that the position at the start of each sequence of instructions for a character is also at this position. Finally, the numbers 3 and 100 must be entered. The 3 signifies the end of the data when it is being used by ECSTASY and the 100 signifies the end of this line of data for program GETCHAR.

For example, the minus (-) sign is:




```
1 4 3   - move to right of the symbol
0 -4 0  - draw to left
1 0 -3  - reposition to start
3 100   - finish off
```

The present implementation is restricted to 23 drawing and moving instructions per letter. This can be changed by increasing the record size used in program GETCHAR but it would mean all the character codes having to be reentered.

7. Picture Displays

The plot files produced by the `send' routine for a specific display device can be interpreted by the appropriate display routine in the following ways.

7.1 Televideo Display

(i) run program TVOUT on the ECLIPSE
(ii) reply to prompt with the name of the plot file
(iii) wait for screen to be cleared and picture displayed

7.2 Printronix Display

(i) Copy the plot file to magnetic tape from the ECLIPSE via the ETOP command. i.e. ETOP MT0:0/o plot_file/i.

(ii) Get the tape mounted on the PRIME, assigning it to MT0.

(iii) Use the MAGNET facility to copy the plot file from tape in the following manner:

```
OK, magnet

[MAGNET rev. 18.1]

OPTION: r
MTU # = 0
MT FILE # = 1 { or whatever }
LOGICAL RECORD SIZE = 514
BLOCKING FACTOR = 1
ASCII, BCD, BINARY, OR EBCDIC? binary
OUTPUT FILE: plotfile
OK,
```

(iv) Load and run program ETOP to preprocess the plot file. The input and output plot file names will be asked for.

(v) Load and run program TOPR0 to produce the final spoolable plot file. The input and output plot file names will be asked for.

(vi) Spool the final plot file to PR0 using the no formatting option. i.e. spool plotfile -at pr0 -nof.

APPENDIX II

ECSTASY System Source Code Listing

```
%  Basic kernel of drawing routines based on the Core
%  Graphics System.

   begin

   integer x_dev_size, y_dev_size,
%             device sizes
           x_text_scale, y_text_scale;
%             text scale factors

   real x_curr_pos, y_curr_pos,
%          current position indicators;
        x_wind_max, y_wind_max,
%          upper window limits
        x_wind_min, y_wind_min,
%          lower window limits
        x_dev_max, y_dev_max,
        x_dev_min, y_dev_min,
%          adjusted device max and mins after viewport and
%          window adjust specifications
        dots_per_x, dots_per_y,
%          device dots per world coord dot
        dot_ratio,
%          x dots per 1 y dot
        x_view_diff, y_view_diff;
%          difference in upper and lower viewport specifications

   procedure get_device( device_code );

%     query user for output device and initialise system params

      integer device_code;
      begin
      character ch;
      if device_code = 1 then begin comment: printronix;
         x_dev_size := 390;
         y_dev_size := 360;
         dot_ratio  := 0.833
         end
      else if device_code = 2 then begin comment: televideo;
         x_dev_size := 79;
         y_dev_size := 21;
         dot_ratio  := 2.35
         end
      else begin comment: device not known;
         outimage;
         outtext( "Output devices:" );     outimage;
         outtext( "  1: Printronix" );     outimage;
         outtext( "  2: Televideo " );     outimage;
                                           outimage;
         outtext( "Enter your choice:" ); outimage;
         inimage;
         ch := inchar;
         get_device( rank(ch) - rank('0') ) comment: recursively;
         end
      end get_device;
```

```
%     get bit map size and initialise system defaults

      begin
      get_device( 0 ); comment: device not yet known
      y_dev_sze := y_dev_size;
      view_port( 0.0, 1.0, 0.0, 1.0 );
      window( 0, x_dev_size, 0, y_dev_size, false );
      text_size(1, 1)
      end y_dev_sze;

   procedure plot( x1, y1 );
%     mark the point (x1, y1) in the bit map

      real x1, y1;
      begin
      integer x, y,
              power;
      if ( x1 >= x_dev_min ) and ( x1 < x_dev_max + 1 )
         and ( y1 >= y_dev_min ) and ( y1 < y_dev_max + 1 )
            then begin
%           point in domain so convert to integer device coords
            x := entier( x1 );
            y := entier( y1 );
            power := mod ( x, 30 );
            power := if power = 0 then 1 comment: for right shift;
                     else 2 ** ( 30 - power );
            x := ( x - 1 ) // 30 + 1; comment: 30 bits per int;
            if mod( bit_map( y, x ) // power, 2 ) = 0 then
%              get right bit and see if is already on
               bit_map( y, x ) := bit_map( y, x ) + power
            end
      end plot;

   procedure join( x1, y1, x2, y2 );
%     mark a line from (x1, y1) to (x2, y2) in device coords

      real x1, y1, x2, y2;
      begin
      integer i;
      real length,
           xincr, yincr;
      length := if abs( y2 - y1 ) > abs( x2 - x1 ) then
         abs( y2 - y1 )
      else abs( x2 - x1 );
      if length = 0 then comment: is actually a point;
         plot( x1, y1 )
      else begin comment: mark the line point by point;
         xincr := ( x2 - x1 ) / length;
         yincr := ( y2 - y1 ) / length;
         for i := 1 step 1 until round( length + 0.5 ) do begin
            plot( x1, y1 );
            x1 := x1 + xincr;
            y1 := y1 + yincr
            end
         end
      end join;
```

```
%       convert world to device coordinates including viewport

        name x, y;
        real x, y;
        begin
        x := x_dev_min + ( x - x_wind_min )
                       * dots_per_x * x_view_diff;
        y := y_dev_min + ( y - y_wind_min )
                       * dots_per_y * y_view_diff
        end map_to_device;

    procedure draw_line( x1, y1, x2, y2 );

%       mark line from (x1, y1) to (x2, y2) in device coords

        real x1, y1, x2, y2;
        begin
        map_to_device( x1, y1 );
        map_to_device( x2, y2 );
%       now in device coords so join
        join( x1, y1, x2, y2 )
        end draw_line;

    procedure reset;

%       clear internal bit map representation of display device

        begin
        integer i, j;
        for i := 1 step 1 until y_dev_size do comment: each line;
           for j := 1 step 1 until ( x_dev_size-1 ) // 30 + 1 do
              bit_map( i, j ) := 0
        end reset;
```

```
%       dump bit map representation to a plot file

        begin
        text title;
        ref ( outfile ) plot_file;
        integer i, j,
                integers;
        title :- blanks(16);
        outimage;
        outtext( "Please enter the output file name:" );
        outimage;
        inimage;
        title := intext(16);
        plot_file :- new outfile( title );
        inspect plot_file do begin
           open( blanks( 84 ) );
           integers := ( x_dev_size - 1 ) // 30 + 1;
%          number of integers for a dot row line
           for i := y_dev_size step -1 until 1 do
%             for each line from the top of the map
              for j := 1 step 1 until integers do begin
                 outint( bit_map( i, j ),
                 outimage
                 end for j;
           close
           end inspect;
        end send;

    procedure window( xlo, xhi, ylo, yhi, adjust );

%       set window specifications and system parameters

        real xlo, xhi, ylo, yhi;
        boolean adjust;
        begin
        x_curr_pos := x_wind_min := xlo;
        x_wind_max := xhi;
        dots_per_x := x_dev_size / ( x_wind_max - x_wind_min );
        y_curr_pos := y_wind_min := ylo;
        y_wind_max := yhi;
        dots_per_y := y_dev_size / ( y_wind_max - y_wind_min );
        if adjust then begin comment: adjust for true scale;
           if dots_per_x > ( dots_per_y * dot_ratio ) then begin
              dots_per_x := dots_per_y * dot_ratio;
              x_dev_max  := (x_wind_max-x_wind_min) * dots_per_x
              end
           else begin
              dots_per_y := dots_per_x / dot_ratio;
              y_dev_max  := (y_wind_max-y_wind_min) * dots_per_y
              end
           end
        end window;
```

```
%       set viewport parameters

        real xlo, xhi, ylo, yhi;
        begin
        x_view_diff := xhi - xlo;
        y_view_diff := yhi - ylo;
        x_dev_max := x_dev_size * xhi;
        x_dev_min := x_dev_size * xlo + 1;
%       device min is atleast 1 and not 0
        y_dev_max := y_dev_size * yhi;
        y_dev_min := y_dev_size * ylo + 1
        end view_port;

    procedure line_abs( x, y );

%       mark a line from CP to (x, y) in world coords

        real x, y;
        begin
        draw_line( x_curr_pos, y_curr_pos, x, y );
        x_curr_pos := x;
        y_curr_pos := y
        end line_abs;

    procedure line_rel( x, y );

%       mark a line from CP to (CP+x, CP+y) in world coords

        real x, y;
        begin
        draw_line( x_curr_pos, y_curr_pos,
                   x_curr_pos+x, y_curr_pos+y );
        x_curr_pos := x_curr_pos + x;
        y_curr_pos := y_curr_pos + y
        end line_rel;

    procedure mark_abs( x, y );
%       mark a point at (x, y) in world coords

        real x, y;
        begin
        x_curr_pos := x;
        y_curr_pos := y;
        map_to_device( x, y );
        plot( x, y )
        end mark_abs;
```

```
%     mark a point at (CP+x, CP+y) in world coords

      real x, y;
      begin
      x_curr_pos := x := x_curr_pos + x;
      y_curr_pos := y := y_curr_pos + y;
      map_to_device( x, y );
      plot( x, y )
      end mark_rel;

   procedure move_abs( x, y );

%     move CP to (x, y) in world coords

      real x, y;
      begin
      x_curr_pos := x;
      y_curr_pos := y
      end move_abs;

   procedure move_rel( x, y );

%     move CP to (CP+x, CP+y) in world coords

      real x, y;
      begin
      x_curr_pos := x_curr_pos + x;
      y_curr_pos := y_curr_pos + y
      end move_rel;

   procedure text_size ( x_size, y_size );

 %    set text scaling factors

      integer x_size, y_size;
      begin
      if ( x_size < 1 ) or ( y_size < 1 ) then begin
         outimage;
         outtext( "**WARNING** Text scale factors must be" );
         outtext( "greater than zero" );
         outimage; outimage
         end
      else begin comment: ok;
         x_text_scale := x_size;
         y_text_scale := y_size
         end
      end text_size;
```

```
    real x, y;
    ref ( directfile ) data;
    begin
    integer command, x_in, y_in;
    data.locate ( rank ( ch ) - 31 );
    data.inimage; comment: get record for the character;
    command := data.inint;
    while command < 2 do begin comment: while not terminator;
       x_in := data.inint;
       y_in := data.inint;
       if command = 0 then comment: is a draw, not a move
          join( x, y, x + ( x_in * x_text_scale ),
                      y + ( y_in * y_text_scale) );
       x := x + x_in * x_text_scale; comment: move CP;
       y := y + y_in * y_text_scale;
       command := data.inint
       end while;
    end letter_draw;

 procedure prOtext ( line, x_start, y_start,
                     x_char_space, y_char_space );

%     mark the string 'line' at (xstart, ystart) in world coords

    value line;
    text line;
    real x_start, y_start
         x_char_space, y_char_space; comment: direction;
    begin
    character ch;
    ref ( directfile ) data; comment: for char codes;
    integer index;
    data :- new directfile ( "charcodes" );
    data.open( blanks (72) );
    map_to_device( x_start, y_start );
    for index := 1 step 1 until line.length do begin
%       for each character in the string
       ch := line.getchar;
       letter_draw ( ch, x_start, y_start, data );
%       now move to start of next char
       x_start := x_start +  7 * x_char_space * x_text_scale;
       y_start := y_start + 10 * y_char_space * y_text_scale
       end for;
    x_curr_pos := x_start;
    y_curr_pos := y_start;
    data.close
    end prOtext;
```

```
%       modulo division

        integer numer, denom;
        begin
        mod := numer - ( numer // denom ) * denom
        end mod;

    integer procedure round( x );

%       rounding function

        real x;
        begin
        round := if x - entier( x ) < 0.5 then entier( x )
                 else entier( x + 1 )
        end round;


    integer array bit_map( i : y_dev_sze,
                           i : ( x_dev_size - 1 ) // 30 + 1 );


    end class plot_environment;
```

```
%  Provide facilities for drawing geometrical primitives

   begin

   head class picture;

%     representation of a complete picture

      begin

      procedure draw;
%        mark the picture elements on the bit map

         begin
         ref (element) item;
         item :- first;
         while item =/= none do begin
%           for each element of the picture
            item.draw;
            item :- item.suc
            end
         end draw;

      ref (picture) procedure rotate( xcentre, ycentre, angle );
%        rotate the complete picture about (xcentre, ycentre)

         real xcentre, ycentre, angle;
         begin
         ref (element) item;
         ref (picture) pix;
         item :- this picture.first;
         pix :- new picture;
         while item =/= none do begin
%           rotate each picture element and add to new picture
            item.rotate( xcentre, ycentre, angle ).into( pix );
            item :- item.suc
            end;
         rotate :- pix
         end rotate;

      ref (picture) procedure scale( xscale, yscale );
%        scale the picture

         real xscale, yscale;
         begin
         ref (element) item;
         ref (picture) pix;
         item :- this picture.first;
         pix :- new picture;
         while item =/= none do begin
%           scale each picture item and add to new picture
            item.scale( xscale, yscale ).into( pix );
            item :- item.suc
            end;
         scale :- pix
         end scale;
```

```
%         translate the picture by x_dir and y_dir units

          real x_dir, y_dir;
          begin
          ref (element) item;
          ref (picture) pix;
          item :- this picture.first;
          pix :- new picture;
          while item =/= none do begin
%             translate each picture item and add to new picture
              item.translate( x_dir, y_dir ).into( pix );
              item :- item.suc
              end;
          translate :- pix
          end translate;

      procedure include( object );

%         add new element to this picture

          ref (element) object;
          begin
          object.into( this picture )
          end include;


      end class picture;

   link class element;

%     super-class so all elements can be referred to by a
%     common reference variable of this class

      virtual : procedure draw;
                ref (element) procedure rotate;
                ref (element) procedure scale;
                ref (element) procedure translate;
      begin
      end;
```

```
%     routines to be applied to circle objects

      ref (point) centre;
      real radius;
      begin
      real x_scale_factor, comment: affected by scaling;
           y_scale_factor;

      procedure draw;

%        draw the circle

         begin
         real x, y,
              circum;
         for circum := 0 step ( 1 / x_dev_size ) until 6.283
            do begin comment: ensure get all points on perimeter;
            x := radius * x_scale_factor
                        * cos( circum ) + centre.x;
            y := radius * y_scale_factor
                        * sin( circum ) + centre.y;
            mark_abs( x, y )
            end for;
         end draw circle;

      ref (circle) procedure rotate ( xcentre, ycentre, angle );
         real xcentre, ycentre,
              angle;
         begin
         ref (circle) circ;
         circ :- new circle( centre, radius );
%        must set scale factors from original
         circ.x_scale_factor := x_scale_factor;
         circ.y_scale_factor := y_scale_factor;
         circ.centre :- centre.rotate( xcentre, ycentre, angle );
         rotate :- circ
         end rotate circle;

      ref (circle) procedure scale ( xscale, yscale );
         real xscale, yscale;
         begin
         ref (circle) circ;
         circ :- new circle( centre, radius );
         circ.x_scale_factor := x_scale_factor * xscale;
         circ.y_scale_factor := y_scale_factor * yscale;
         circ.centre :- centre.scale( xscale, yscale );
         scale :- circ
         end scale circle;
```

```
    begin
    ref (circle) circ;
    circ := new circle( centre, radius );
    circ.x_scale_factor := x_scale_factor;
    circ.y_scale_factor := y_scale_factor;
    circ.centre := centre.translate( x_dir, y_dir );
    translate := circ
    end translate circle;

x_scale_factor := y_scale_factor := 1

end element class circle;
```

```
begin

procedure draw;
   begin
   mark_abs( x, y )
   end draw;

ref (point) procedure rotate( xcentre, ycentre, angle );
   real xcentre, ycentre, angle;
   begin
   real tx, ty;
   ref (point) pt;
   pt :- new point ( 0, 0 );
   tx := x - xcentre;
   ty := y - ycentre;
   pt.x := tx * cos( angle ) - ty
               * sin( angle ) + xcentre;
   pt.y := tx * sin( angle ) + ty
               * cos( angle ) + ycentre;
   rotate :- pt
   end rotate;

ref (point) procedure scale( xscale, yscale );
   real xscale, yscale;
   begin
   ref (point) pt;
   pt :- new point ( 0, 0 );
   pt.x := x * xscale;
   pt.y := y * yscale;
   scale :- pt
   end scale;

ref (point) procedure translate( x_dir, y_dir );
   real x_dir, y_dir;
   begin
   ref (point) pt;
   pt :- new point( 0, 0 );
   pt.x := x + x_dir;
   pt.y := y + y_dir;
   translate :- pt
   end translate;

end class point;
```

```
    ref (head) chain;
%   for linked list of points that define the polygon

    procedure define( co_ords, number );

%       define the polygon from `number' points in array co_ords

        real array co_ords;
        integer number;
        begin
        integer index;
        ref (point) pt;
        for index := 1 step 1 until number do begin
%           for each point in the array create an instance
            pt :- new point( co_ords( index, 1 ),
                             co_ords( index, 2 ) );
%           and add to polygon list
            pt.into( chain )
            end for;
        end define;

    procedure include( pt );

%       include a single point in the polygon representation

        ref ( point ) pt;
        begin
        pt.into( chain )
        end include;

    procedure draw;
        begin
        ref (point) item;
        item :- chain.first;
        if item =/= none then begin
%           move CP to first point in list
            move_abs( item.x, item.y );
            item :- item.suc
            end if;
        while item =/= none do begin
%           draw a line to the next point
            line_abs( item.x, item.y );
            item :- item.suc
            end while;
        item :- chain.first;
        line_abs( item.x, item.y )
        end draw;
```

```
        begin
        ref (point) pt;
        ref (polygon) poly;
        poly :- new polygon;
        pt :- chain.first;
        while pt =/= none do begin
%           rotate each point and add to new polygons list
            pt.rotate( xcentre, ycentre, angle ).into( poly.chain );
            pt :- pt.suc
            end;
        rotate :- poly
        end rotate;

    ref (polygon) procedure scale( xscale, yscale );
        real xscale, yscale;
        begin
        ref (point) pt;
        ref (polygon) poly;
        poly :- new polygon;
        pt :- chain.first;
        while pt =/= none do begin
%           scale each point and add to new polygons list
            pt.scale( xscale, yscale ).into( poly.chain );
            pt :- pt.suc
            end while;
        scale :- poly
        end scale;

    ref (polygon) procedure translate( x_dir, y_dir );
        real x_dir, y_dir;
        begin
        ref (point) pt;
        ref (polygon) poly;
        poly :- new polygon;
        pt :- chain.first;
        while pt =/= none do begin
%           translate each point and add to new polygons list
            pt.translate( x_dir, y_dir ).into( poly.chain );
            pt :- pt.suc
            end while;
        translate :- poly
        end translate;

    chain :- new head;
%   initialise chain on each new polygon creation

    end class polygon;
```

```
%     line is just a special case of polygon with two points

      real x1, y1, x2, y2;
      begin
      ref (point) pt;
%     include two points into polygon representation
      pt :- new point( x1, y1 );
      include( pt );
      pt :- new point( x2, y2 );
      include( pt )
      end class line ;
```

```
%     read object codes and data off data file for 'number' objects,
%     create new instances and add to picture referenced by pix.

      value filnam;
      name pix;
      text filnam;
      integer number;
      ref ( picture ) pix;
      begin
      ref ( infile ) data;
      ref ( point ) pt;
      ref ( circle ) circ;
      ref ( polygon ) poly;
      ref ( line ) lin;
      character ch;
      integer i, j,
              marks;

      data :- new infile( filnam );
      data.open( blanks(127) );
      for i := 1 step 1 until number do begin
%        for each object create instance and add to picture
         data.inimage;
         ch := data.inchar;
         if ( ch = 'c' ) or ( ch = 'C' ) then begin
            comment: circle;
            pt :- new point ( data.inreal, data.inreal );
            circ :- new circle ( pt, data.inreal );
            pix.include( circ )
            end
         else if ( ch = 'p' ) or ( ch ='P' ) then begin
            comment; point;
            pt :- new point ( data.inreal, data.inreal );
            pix.include( pt )
            end
         else if ( ch = 'g' ) or ( ch = 'G' ) then begin
            comment: polygon;
            poly :- new polygon;
            marks := data.inint;
            for j:= 1 step 1 until marks do begin
               pt :- new point ( data.inreal, data.inreal );
               poly.include( pt )
               end;
            pix.include( poly )
            end
         else if ( ch = 'l' ) or ( ch = 'L' ) then begin
            comment: line;
            lin :- new line ( data.inreal, data.inreal,
                              data.inreal, data.inreal );
            pix.include( lin )
            end
```

```
            outtext("**WARNING** unrecognized object code in file ");
            outtext( filnam );
            outimage;   outimage
            end
        end;
    data.close
    end get_objects;

end class animation;
```

APPENDIX III

Display Routines

Contains:

(i) Televideo Display Routine
(ii) ETOP program for preprocessing of Printronix plot file
(iii) Printronix Display Routine

Program TVOUT

```
% A program to interpret the data in the given plot file and
% display the represented image on a Televideo terminal.
begin
integer last_bit,
        int,           comment: integer from plot file;
        bit,           comment: last bit extracted;
        right_shift,   comment: for right shift on integer;
        lines,         comment: loop controls;
        integers,
        bits;
text title;
ref ( infile ) plot_file;
title :- blanks( 16 );
outtext( "Please enter the name of the plot file: " );
outimage;
inimage;
title := intext(16);
plot_file :- new infile ( title );
plot_file.open( blanks(36) );
outchar( char(26) );
outimage;
for lines := 1 step 1 until 21 do begin
   for integers := 1 step 1 until 3 do begin
      plot_file.inimage;
      int := plot_file.inint;
      last_bit := if 30 * integers - 79 <= 0 then 0
                  else 30 * integers - 79;
      comment: last_bit ensures all bits in last int not used;
      if int > 0 then begin
         for bits := 29 step -1 until last_bit do begin
            comment: extract bit from integer;
            right_shift := 2 ** bits;
            bit := int // right_shift;
            bit := bit - 2 * ( bit // 2 );
            if bit = 1 then
               outtext( "#" )
            else
               outtext( " " )
            end
         end
      else
         for bits := 29 step -1 until last_bit do
            outtext( " " )
      end for integers;
   outimage
   end for lines;
plot_file.close
end program;
```

```
program etop ( infile, outfile ); {
============

  This program is used to preprocess the plot file for
  the Printronix before it is used as data to the display
  routine for this device }

var
   ch : char;
   filnam : packed array [ 1 .. 16 ] of char;
   infile, outfile : text;

begin
writeln;
write( 'Enter name of file to be fixed: ');
readln( filnam );
reset( infile, filnam );
writeln;
write( 'Enter name of output file: ');
readln( filnam );
rewrite( outfile, filnam );
writeln;
writeln( 'Begin processing ... ' ); { be friendly }
writeln;
while not eof(infile) do begin
   { get rid of all unwanted characters from the plot file }
   read(infile, ch );
   if ord(ch) >= ord(' ') then write( outfile, ch ) { ok }
   else if ord(ch) = 13 then writeln(outfile) { a linefeed }
   end;
writeln(outfile)
end { etop }.
```

```
program plotfile_for_pr0 ( indata, outdata ); {
=============================

  This is the display routine for the Printronix printer. }

const

    lines_per_page     = 360;
    integers_per_line =  13;   { this many to represent 390 dots }
    chars_per_int      =  10;   { plot characters per integer }
    bits_per_char      =   3;   { three bits for each plot character }
    rows_per_line      =   2;   { double up on dot size }
    chars_per_line     = 130;
    plot_mode          =   5;   { ASCII character codes }
    form_feed          =  12;

var

    line   : packed array [ 1 .. chars_per_line ] of char;
    infile : packed array [ 1 .. 16 ] of char;

    column,        { which char on a dot row we are at }
    three_bits,    { value of last three bit string  extracted }
    right_shift,   { divisor for performing a right shift }
    int,           { integer read from plot file }
    modulo,        { two to the power of bits_per_char }
    integers,      { loop controls }
    chars,
    lines,
    rows         : integer;

    indata,
    outdata  : text;  { input and output plot files }

function power ( base,
                 exponent : integer ) : integer;

{ for exponentiation }

var
    loop,
    temp : integer;

begin
temp := 1;
for loop := 1 to exponent do
    temp := temp * base;
power := temp
end { power };
```

```
begin { main }
writeln;
write( 'Enter input  plot file name :  ' );
readln( infile );
reset( indata, infile );
writeln;
write( 'Enter output plot file name :  ' );
readln( infile );
rewrite( outdata, infile );
writeln;
writeln( 'Processing begins ...  ' ); { politeness }

writeln( outdata, chr(form_feed) ); { new page as no formatting }
modulo := power( 2, bits_per_char ); { for bit extraction later }
for lines := 1 to lines_per_page do begin
   column := 0;
   for integers := 1 to integers_per_line do begin
      read( indata, int );
      if int > 0 then { skip all this calculation }
         for chars := ( chars_per_int - 1 ) downto 0 do begin
            { get value represented by each 3 bit string in the integer }
            right_shift := power( 2, chars * bits_per_char );
            three_bits := ( int div right_shift ) mod modulo;
            column := column + 1;
            case three_bits of { send appropriate plot character }
               0 : line[ column ] := chr( 192 );
               1 : line[ column ] := chr( 240 );
               2 : line[ column ] := chr( 204 );
               3 : line[ column ] := chr( 252 );
               4 : line[ column ] := chr( 195 );
               5 : line[ column ] := chr( 243 );
               6 : line[ column ] := chr( 207 );
               7 : line[ column ] := chr( 255 )
               end { case }
            end { for chars }
      else
         for chars := ( chars_per_int -1 ) downto 0 do begin
            column := column + 1;
            line[ column ] := chr( 192 )
            end { for chars }
      end { for integers };
   for rows := 1 to rows_per_line do { send 2 plot data lines }
      writeln( outdata, line, chr( plot_mode ) )
   end { for lines };

writeln;
writeln( ' ... and successfully terminates.' );
writeln;
close( outdata );
close( indata )
end { main }.
```

APPENDIX IV

Character Codes

Contains:

(i) Program GETCHAR for input of character codes

(ii) Coded character representations

Program GETCHAR

```
% A program which allows the user to alter the coded character
% representations stored in file CHARCODES.


begin
character ch;
integer val;
ref ( directfile ) data;
data :- new directfile( "charcodes" );
data.open( blanks(72) );
ch := 'a';
while ( ch <> 'y' ) and ( ch <> 'Y' ) do begin
%  get next character representation;
   outtext( "Enter the character: " );
   outimage;  inimage;
   ch := inchar;
   val := rank( ch ) - 31; comment: get direct file address
   data.locate( val );
   outtext( "Enter plotting code for the character: " );
   outimage;  inimage;
   val := ininT;
   while val <> 100 do begin
%     user inputs value of 100 to signify end of input
      data.outint( val, 2 );
      val := inint
      end;
   data.outchar( ' ' );
   data.outchar(ch);  comment: output char that is represented;
   data.outimage;
   outtext( "Type 'y' to stop: " );
   outimage;  inimage;
   ch := inchar;
   end while;
data.close
end getchar;
```

This is a listing of file CHARCODES. It contains the coded character representations used by the text facility associated with the ECSTASY Graphics System.

```
3
1 2 6 0 0-4 1 0-2 0 0 0 1-2 0 3 !
1 1 6 0 0-1 1 2 1 0 0-1 1-3-5 3 "
1 1 6 0 0-6 1 2 6 0 0-6 1 1 4 0-4 0 1 4-2 0-4 0 1 0-2 3 #
0 3 0 0 1 1 0-4 4 0 1 1 0 3 0 1-2 0 0 0-6 1-2 0 3 $
1 1 5 0 0 0 1 3 0 0-4-4 1 3 0 0 0 0 1-3-1 3 %
1 4 0 0-4 4 0 0 1 0 1 1 0 2 0 0 1-1 0 0-1 0-3-3 0 0-1 0 3 0 1-4 0 3 &
1 2 6 0 0-2 1-2-4 3 '
1 2 6 0-2-2 0 0-2 0 2-2 1-2 0 3 (
1 2 6 0 2-2 0 0-2 0-2-2 1-2 0 3 )
1 0 5 0 4-4 1 0 4 0-4-4 1 2 4 0 0-4 1-2-1 3 *
1 2 5 0 0-4 1 2 2 0-4 0 1 0-3 3 +
1 1-2 0 1 1 0 0 2 0-1 0 0 0-1 0 1 0 1-2 0 3 ,
1 4 3 0-4 0 1 0-3 3 -
1 1 0 0 0 1 0 1 0 0 0-1 0-1 0 1-1 0 3 .
1 4 6 0 0-1 0-4-4 0 0-1 3 /
1 1 0 0 2 0 0 1 1 0 0 4 0-1 1 0-2 0 0-1-1 0 0-4 0 1-1 1-1 0 3 0
0 4 0 1-2 0 0 0 6 0-1 0 0-1-1 1 0-5 3 1
1 4 0 0-4 0 0 0 1 0 2 2 0 1 0 0 1 1 0 0 1 0-1 1 0-2 0 0-1-1 1 0-5 3 2
0 3 0 0 1 1 0 0 1 0-1 1 0-1 0 0 1 0 0 1 1 0 0 1 0-1 1 0-3 0 1 0-6 3 3
1 3 0 0 0 6 0-3-3 0 0-1 0 4 0 1-4-2 3 4
1 0 1 0 1-1 0 2 0 0 1 1 0 0 2 0-1 1 0-3 0 0 0 2 0 4 0 1-4-6 3 5
1 4 6 0-3 0 0-1-1 0 0-4 0 1-1 0 2 0 0 1 1 0 0 1 0-1 1 0-3 0 1 0-3 3 6
0 4 4 0 0 2 0-4 0 1 0-6 3 7
0 4 0 0 0 6 0-4 0 0 0-3 0 4 0 1-4 0 0 0-3 3 8
1 4 0 0 0 5 0-1 1 0-2 0 0-1-1 0 0-1 0 1-1 0 3 0 1-4-3 3 9
1 2 3 0 0 0 1 0-2 0 0 0 1-2-2 3 :
1 2 3 0 0 0 1 0-2 0 0-1 0-1-1 1-1 1 3 ;
1 4 6 0-3-3 0 3-3 1-4 0 3 <
1 4 4 0-4 0 1 4-2 0-4 0 1 0-2 3 =
1 0 6 0 3-3 0-3-3 3 >
1 0 5 0 1 1 0 2 0 0 1-1 0 0-1 0-2-2 1 0-2 0 0 0 1-2 0 3 ?
1 0 5 0 1 1 0 2 0 0 1-1 0 0-4 0-1-1 0-3 0 0 0 3 0 2 0 0 0-3 1-2 0 3 @
0 0 4 0 2 2 0 2-2 0 0-4 1 0 2 0-4 0 1 0-2 3 A
0 3 0 0 1 1 0 0 1 0-1 1 0-1 0 1 1 0 0 1 1 0 0 1 0-1 1 0-3 0 0 0-6 3 B
1 4 1 0-1-1 0-2 0 0-1 1 0 0 4 0 1 1 0 2 0 0 1-1 1-4-5 3 C
0 3 0 0 1 1 0 0 4 0-1 1 0-3 0 0 0-6 3 D
1 4 0 0-4 0 0 0 6 1 4 0 0-4 0 1 2-3 0-2 0 1 0-3 3 E
0 0 6 0 4 0 1-2-3 0-2 0 1 0-3 3 F
1 3 2 0 1 0 0 0-1 0-1-1 0-2 0 0-1 1 0 0 4 0 1 1 0 2 0 0 1-1 1-4-5 3 G
1 0 6 0 0-6 1 0 3 0 4 0 1 0 3 0 0-6 1-4 0 3 H
1 4 0 0-4 0 1 2 0 0 0 6 1 2 0 0-4 0 1 0-6 3 I
1 4 6 0 0-5 0-1-1 0-2 0 0-1 1 0 0 1 1 0-2 3 J
1 0 6 0 0-6 1 4 6 0-3-3 0 3-3 1-4 0 3 K
1 0 6 0 0-6 1 4 0 0-4 0 3 L
0 0 6 0 2-2 0 0-1 1 0 1 0 2 2 0 0-6 1-4 0 3 M
1 0 6 0 0-6 1 0 5 0 4-4 1 0 5 0 0-6 1-4 0 3 N
1 1 0 0 2 0 0 1 1 0 0 4 0-1 1 0-2 0 0-1-1 0 0-4 0 1-1 1-1 0 3 0
```

```
0 0 6 0 3 0 0 1-1 0 0-1 0-1-1 0-3 0 1 0-3 3 P
1 1 0 0 3 0 0 0 5 0-1 1 0-2 0 0-1-1 0 0-4 0 1-1 1 1 2 0 3-3 1-5 1 3 Q
1 4 0 0 0 2 0-1 1 0-3 0 1 3 0 0 1 1 0 0 2 0-4 0 0 0-6 3 R
0 3 0 0 1 1 0 0 1 0-1 1 0-2 0 0-1 1 0 0 1 0 1 1 0 3 0 1-4-6 3 S
1 2 0 0 0 6 1 2 0 0-4 0 1 0-6 3 T
1 0 6 0 0 0-5 0 1-1 0 1 0 0 2 2 1 0 4 0 0-6 1-4 0 3 U
1 0 6 0 0 0-4 0 2-2 1 2 6 0 0-4 0-2-2 1-2 0 3 V
1 0 6 0 0 0-5 0 1-1 0 1 1 0 0 1 1 0-1 0 1-1 0 1 1 1 0 5 0 0-5 1-4-1 3 W
1 0 6 0 2-2 0 0-2 0-2-2 1 4 6 0-2-2 1 0-2 0 2-2 1-4 0 3 X
1 0 6 0 2-2 1 2 2 0-2-2 0 0-4 1-2 0 3 Y
1 4 0 0-4 0 0 0 1 0 4 4 0 0 1 0-4 0 1 0-6 3 Z
1 4 6 0-4 0 0 0-6 1 4 0 0-4 0 3 [
1 0 6 0 4 0 0 0-6 0-4 0 3 ]
1 0 4 0 2 2 0 2-2 1-4-4 3 ^
1 4 0 0-4 0 3
1 2 6 0 2-2 1-4-4 3 `
1 1 4 0 3 0 0 0-4 0-4 0 0 0 2 0 4 0 1-4-2 3 a
1 0 6 0 0-6 0 3 0 0 1 1 0 0 2 0-1 1 0-3 0 1 0-4 3 b
1 4 4 0-3 0 0-1-1 0 0-2 0 1-1 0 3 0 1-4 0 3 c
1 4 6 0 0-6 0-3 0 0-1 1 0 0 2 0 1 1 0 3 0 1-4-4 3 d
1 4 0 0-4 0 0 0 4 0 4 0 0 0-2 0-4 0 1 0-2 3 e
1 1 0 0 0 5 0 1 1 0 1 0 0 1-1 1-2-2 0-2 0 1 0-3 3 f
0 4 0 0 0 4 0-4 0 0 0-2 0 4 0 1-4-2 3 g
1 0 6 0 0-6 1 0 4 0 4 0 0 0-4 1-4 0 3 h
1 2 6 0 0 0 1-1-2 0 1 0 0 0-4 1 2 0 0-4 0 3 i
1 3 6 0 0 0 1 0-2 0 0-3 0-1-1 0-1 0 0-1 1 0 0 1 1 0-2 3 j
1 0 6 0 0-6 1 4 4 0-2-2 0 2-2 1-2 2 0-2 0 1 0-2 3 k
1 1 6 0 1 0 0 0-6 1 2 0 0-4 0 3 1
0 0 3 0 1 1 0 1-1 0 0-3 1 0 3 0 1 1 0 1-1 0 0-3 1-4 0 3 m
0 0 3 0 1 1 0 2 0 0 1-1 0 0-3 1-4 0 3 n
1 1 0 0 2 0 0 1 1 0 0 2 0-1 1 0-2 0 0-1-1 0 0-2 0 1-1 1-1 0 3 o
0 0 4 0 3 0 0 1-1 0-1-1 0-3 0 1 0-2 3 p
1 4 0 0 0 3 0-1 1 0-2 0 0-1-1 0 1-1 0 3 0 1-4-2 3 q
0 0 3 0 1 1 0 2 0 0 1-1 1-4-3 3 r
1 4 4 0-3 0 0-1-1 0 1-1 0 2 0 0 1-1 0-1-1 0-3 0 3 s
1 4 4 0-3 0 1 1 2 0 0-5 0 1-1 0 1 1 1-4-1 3 t
1 0 4 0 0-3 0 1-1 0 1 0 0 2 2 1 0 2 0 0-4 1-4 0 3 u
1 0 4 0 0-2 0 2-2 1 2 4 0 0-2 0-2-2 1-2 0 3 v
1 0 4 0 0-3 0 1-1 0 1 1 1 2 3 0 0-3 0-1-1 0-1 1 1-2-1 3 w
1 0 4 0 4-4 1 0 4 0-4-4 3 x
1 0 4 0 0-2 0 4 0 1 0 2 0 0-4 0-4 0 3 y
1 4 0 0-4 0 0 4 4 0-4 0 1 0-4 3 z
1 4 6 0-2 0 0-1-1 0 0-4 0 1-1 0 2 0 1-3 3 0-1 0 1 0-3 3 {
1 2 6 0 0-6 1-2 0 3 |
1 0 6 0 2 0 0 1-1 0 0-4 0-1-1 0-2 0 1 4 3 0-1 0 1-3-3 3 }
1 4 6 0-1-1 0-1 0 0-1 1 0-1-1 1 0-5 3 ~
```

APPENDIX V

Graphics Examples

Contains:

(i) Program EXAMPLE, associated data files and output
(ii) Program EXAMPLE2 and output

Program EXAMPLE

This program, using the data on the following two pages, produces figures a. and b. which follow.

```
begin
%include ECSTASY
animation
    begin
    ref ( picture ) pix;
    ref ( circle ) circ;
    ref ( point ) pt;
    window( 0, 390, 0, 360, false );
    pix :- new picture;
    get_objects( "dataone", 5, pix ); comment: first picture data;
    prOtext( "I'm in ECSTASY!!", 140, 115, 1, 0 );
    pix.draw;
    send;       comment: produce plot file for first picture;
    reset;      comment: reset for next picture;
    window( 0, 390, 0, 360, true ); comment: true to scale;
    pix :- new picture;
    pt :- new point( 310, 300 );
    circ :- new circle( pt, 20 );
    circ.x_scale_factor := 2;    comment: an ellipse;
    pix.include( circ );
    get_objects( "datatwo", 7, pix ); comment: second picture data;
    prOtext( "You can:", 20, 340, 1, 0 );
    prOtext( "Draw circles? -", 40, 300, 1, 0 );
    prOtext( "Shapes -", 40, 240, 1, 0 );
    prOtext( "And have text of:", 40, 170, 1, 0 );
    prOtext( "different s", 60, 130, 1, 0 );
    text_size( 2, 2 );
    prOtext( "IZE", 150, 130, 1, 0 );
    text_size( 1, 1 );
    prOtext( "s and", 198, 130, 0, -1 );
    prOtext( "irections", 208, 100, 1, 1 );
    prOtext( "with all the characters - * / @ ( ) etc.", 40, 50, 1, 0 );
    pix.draw;
    send      comment: plot file for second picture;
    end
end;
```

Data file DATAONE used by program EXAMPLE.

```
C 195 180 160
G 8 115 100 275 100 315 140 295 160 275 140 115 140 95 160 75 140
G 10 185 180 175 160 185 150 205 150 215 160 205 180 205 230 200
     240 190 240 185 230
G 8 135 260 155 220 175 260 155 300 135 260 155 240 175 240 155 280
G 8 215 260 235 220 255 260 235 300 215 260 235 240 255 260 235 280
```

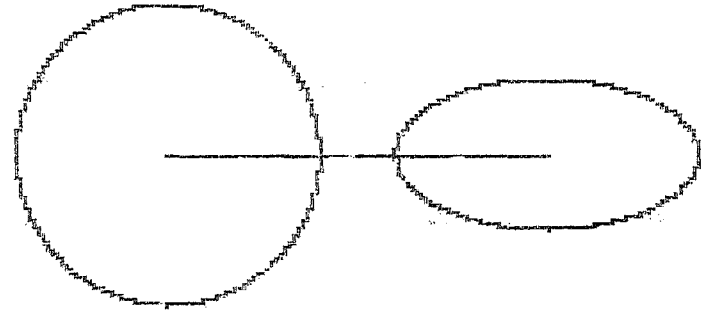
Data file DATATWO used by program EXAMPLE.

```
C 210 300 40
L 210 300 310 300
L 132 204 228 204
L 148 220 212 220
G 4 152 224 208 224 208 244 152 244
G 4 148 212 212 212 208 216 152 216
G 8 212 220 212 248 148 248 148 220 132 204 132 196
    228 196 228 204
```

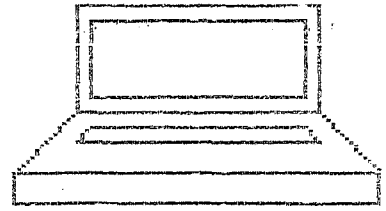
Figure a.

You can:

Draw circles? -

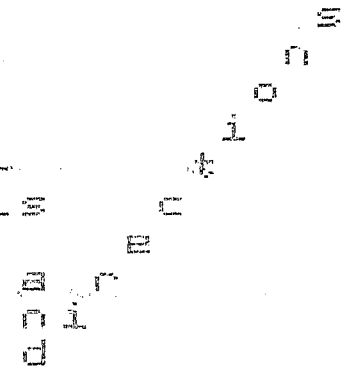


Shapes -



And have text of:

different sIZEs



with all the characters - * / @ ( ) etc.

Figure b.

Program EXAMPLE2

The picture on the following page was produced by this program.

```
begin
%include ecstasy
animation
    begin
    ref (polygon) poly;
    ref (picture) pix;
    real array p( 1:4, 1:2 ); comment: for storing polygon data;
    window( 0, 80, 0, 22, false );
    p(1,1) := 5 ; p(1,2) := 18; p(2,1) := 15;
    p(2,2) := 15; p(3,1) := 20; p(3,2) := 18;
    p(4,1) := 10; p(4,2) := 21;
    pix :- new picture;
    poly :- new polygon;
    poly.define( p, 4 );
    pix.include( poly );
    poly :-  poly.translate( 25, 0 );
    pix.include( poly );
    poly :- poly.scale( 1.5, 1.5 );
    poly :- poly.translate( -20.0, -19.5 );
    pix.include( poly );
    pix.include( poly.rotate( 25, 7.5, 3.1415926 ));
    pix.draw;
    prOtext( "1. Original", 5, 14, 1, 0 );
    prOtext( "2. After translation", 30, 14, 1, 0 );
    prOtext( "3. And Scaling", 30, 2, 1, 0 );
    prOtext( "4. And rotation", 5, 2, 1, 0 );
    send
    end
end;
```

1. Original
2. Rotation
3. Rotation and scaling
4. Rotation and scaling

Figure c.